**CS29003 ALGORITHMS LABORATORY**
**Practice Assignment**
**Date: 10 – Sep – 2020**

# Designing a Reverse-Polish Notation Calculator

## Exercise 1: Convert the given arithmetic expression into Reverse-Polish Notation

Arithmetic expressions are made of operators (+, *, etc.) and operands. The operands may be numbers, variables or (recursively) smaller arithmetic expressions. The expressions can be written in a variety of notations. The most commonly used notation is the one where the operator is written between its operands. Such a notation requires notions of ***precedence*** (indicating, for example, that division has higher precedence than addition, so that 3+15/3 is 8, not 6) and ***associativity*** (indicating, for example, that subtractions are to be done from left to right, so that 10-3-2 is 5, not 9). It also normally requires a way to override the precedence rules by using ***parentheses***. For example, expression 7+3*6 has value 25, but (7+3)*6 has value 60.

An alternative notation is the **Reverse-Polish** notation, where the operator is written after both of its operands. For example, you write 7 56 + to indicate the result of adding 7 and 56. Reverse-Polish notation does not require any notions of precedence or associativity, and does not require parentheses. For example, an arithmetic expression 7+(32*6) is equivalent to Reverse-Polish expression 7 32 6 * +, but the expression (7+32)*6 is equivalent to Reverse-Polish expression 7 32 + 6 *.

In this exercise, you are required to write a program that reads a file titled “input.txt” containing an arithmetic expression on each line written in commonly used notation, converts it into its corresponding Reverse-Polish notation using an ***Operator Stack*** and writes the converted notation on each line in a file titled “part1-output.txt”. The expressions that your program needs to handle involve (decimal) integer constants, such as 4 and 57 and operators as detailed in Table 1. **Make sure to include space between tokens (constants/operators) while writing the converted output into the file** in order to ensure that expressions like 4+57 and 45+7 do not have the same Reverse-Polish Notation 457+.

| **Token** | **Operator** | **Precedence** | **Associativity** |
|---|---|---|---|
| ( ) | parentheses | 3 | left-to-right |
| * / % | multiplication, division, modulo | 2 | left-to-right |
| + - | addition, subtraction | 1 | left-to-right |

Table 1: Operator Precedence and Associativity

## Algorithm and Templates

1. Create an Operator Stack and its associated functions.
   **struct Stack* createStack(int length_of_given_arithmetic_expression);**
   **int isEmpty(struct Stack* stack);**
   **char peek(struct Stack* stack);**
   **char pop(struct Stack* stack);**
   **void push(struct Stack* stack, char op);**

2. Scan the given arithmetic expression from left to right. For each token t (multi-digit operands like *72, 45* etc. are to be considered) in the input string:
   (a) if (t is an operand): append t onto the end of the output string.
   **int isOperand(char* ch);**
   (b) else if (t is an operator):
      i. Pop tokens (if any) from the top of the stack while they have equal or higher precedence than t and append them onto the end of the output string. You can also stop if the top of the stack is an open parenthesis, or if the stack is empty.
      **int precedence(char op1, char op2);**
      ii. Push t into the stack.
   (c) else if (t is a left parenthesis): Push t into the stack.
   (d) else if (t is a right parenthesis):
      i. Pop tokens from the top of the stack and append them onto the end of the output string until a left parentheses is encountered.
      ii. Pop and discard the left parentheses from the stack.

For each line in the input file titled "input.txt", scan the contents (an arithmetic expression) into a character string "exp", call a function with the following template to convert it into Reverse-Polish notation and finally write it to "part1-output.txt". **void convert(char* exp);**

## Exercise 2: Evaluate the arithmetic expressions in Reverse-Polish Notation

In this exercise, you are required to write a program that simulates a Reverse-Polish Calculator. More specifically, the program should take, as input, an expression in Reverse-Polish notation, and return, as output, the computed value of the given expression using an ***Operand Stack***.

## Algorithm and Templates

1. Create an Operand Stack and its associated functions similar to the previous exercise; only difference being that here you would push operands (integer constants) into the stack instead of operators (characters).
2. Scan the arithmetic expression in Reverse-Polish notation from left to right. For each token t (separated by "space") in the input string:
   (a) if (t is an operand): Push t into the stack.
   (b) else if (t is an operator):
      i. Pop the top two operands from the stack.
      ii. Apply the operator t to the two popped operands.
      iii. Push the result of the operation into the stack.
3. When there are no tokens left, the stack only contains one item: the final value of the expression. Pop the stack one final time and write the result into the output file.

For each line in "part1-output.txt", scan the arithmetic expression in Reverse-Polish notation into a character string "exp", call a function with the following template to evaluate it and finally write the obtained value to "part2-output.txt". **long evaluate(char* exp);**

## Sample Input - ”input.txt”

```
3*7+5
45/3*5
56*(3+72)
11%3-2
```

## Sample Output - ”part1-output.txt”

```
3 7 * 5 +
45 3 / 5 *
56 3 72 + *
11 3 % 2 -
```

## Sample Output - ”part2-output.txt”

```
26
75
4200
0
```